

Bounded Model Checking of Temporal Formulas with Alloy

Alcino Cunha

`alcino@di.uminho.pt`

HASlab / INESC TEC and Universidade do Minho

July 12, 2012

Abstract

Alloy is formal modeling language based on first-order relational logic, with no specific support for specifying reactive systems. We propose the usage of temporal logic to specify such systems, and show how bounded model checking can be performed with the Alloy Analyzer.

1 Introduction

Alloy is a formal language based on first-order relational logic [7]. It is supported by a SAT solver that enables model validation and verification. Alloy's logic is quite generic and does not commit to a particular specification style. In particular, there is no standard way to model and verify reactive systems, and several idioms and extensions have been proposed to address this issue.

The local state idiom is one of the most popular to model and reason about operations and execution traces. However, it is rather cumbersome and error-prone to specify safety and liveness properties over such traces. In this report we propose the usage of standard temporal logic LTL to specify such properties, and show how LTL bounded model checking can be performed with the Alloy Analyzer, by resorting to the technique first proposed in [1].

This report is structured as follows. The local state idiom will be described in the next section, together with a brief presentation of Alloy. Section 3 shows how reactive systems can be modelled and specified currently in Alloy, and discusses some of the issues that arise when model checking temporal properties using the Alloy Analyzer. Section 4 discusses how such properties can be specified more easily in LTL and then translated to Alloy to perform safe (bounded) model-checked with the Alloy Analyzer. We discuss some related work in Section 5 and conclude in Section 6.

2 Alloy and the local state idiom

In Alloy, a *signature declaration* represents a set of atoms. An atom is a unity with three fundamental properties: it is indivisible, immutable and uninterpreted. Optionally, a signature declaration can introduce *fields*. Fields represent sets of tuples of atoms and are interpreted as relations between signatures. Model constraints are defined by *facts*. *Assertions* express properties that are expected to hold as consequence of the stated facts. *Commands* are instructions to perform particular analysis. Alloy provides two commands for analysis: **run** and **check**. Command **run** instructs the analyzer to search for an instance satisfying a given formula, and **check** attempts to contradict a formula by searching for a counterexample.

Typically operations are modeled as predicates that specify the relationship between pre- and post-states. Two variants of this idiom are possible, known respectively as *global state* and *local state*. In the former all mutable fields are defined in a global state signature. In the later, the state signature is added locally as an extra column at the end of each mutable field. The local state idiom is more modular, since fields are still declared in the signature they naturally belong to. On the other hand, the global state idiom forces all mutable fields to be artificially grouped together. Without loss of generality, we will also assume the distinguished state signature to be denoted as **Time**. A more formal description of the local state idiom can be found in [5].

Figure 1 presents an example of an Alloy model conforming to the local state idiom. It is a simplified model of the *Partition Information Flow Policy* (PIFP) of a *Secure Partitioning Kernel* (SPK) [3]. Essentially, the PIFP statically defines which information flows between partitions are authorized. Here we assume all information flows to be triggered by message passing.

Signature **Message** declares the attributes **from**, and **to**, which are functional binary relations denoting the single source and destination **Partition**, respectively. Communication is done via channels, here simplified to contain sets of messages. The messages contained in a channel vary over time. As such, signature **Channel** declares a mutable relation **messages** that, for each channel and time point returns the set of messages contained in the channel at that time. Since we are only interested in modeling the PIFP, signature **Partition** just declares two functional binary relations: **kernel** stating in which kernel it runs, and **port** that states which channel will be used by the partition to communicate. Finally, signature **Kernel** declares a ternary relation **pifp**, that for a given kernel and partition determines the partitions it is authorized to send messages.

In Alloy *everything is a relation*. For example, sets (namely signatures) are unary relations and variables are just unary singleton relations. As such, the relational composition operator can be used for various purposes. For example, in the fact **NoSharedChannels** the relational expression **port.c** denotes the set of partitions with port **c**. Multiplicities are also used in several contexts to constrain or check the cardinality of a relation. For example, in the fact **NoSharedChannels**, multiplicity **lone** ensures that each channel is the port of

```

sig Time {}

sig Message { to, from : one Partition }

sig Channel { messages : Message set -> Time }

sig Partition { kernel : one Kernel, port : one Channel }

fact NoSharedChannels { all c : Channel | lone port.c }

sig Kernel { pifp : Partition -> set Partition }

fact PifpOk { all k : Kernel | k.pifp in kernel.k -> kernel.k }

pred send [i, o : Partition, t, t' : Time] {
  some m : from.i & to.o | i.port.messages.t' = i.port.messages.t + m
  all p : Partition - i | p.port.messages.t' = p.port.messages.t
}

pred receive [o : Partition, t, t' : Time] {
  some m : o.port.messages.t & to.o | o.port.messages.t' = o.port.messages.t - m
  all p : Partition - o | p.port.messages.t' = p.port.messages.t
}

pred transfer [k : Kernel, t, t' : Time] {
  some m : (kernel.k).port.messages.t {
    m.to in (m.from).(k.pifp)
    m.from.port.messages.t' = m.from.port.messages.t - m
    m.to.port.messages.t' = m.to.port.messages.t + m
    all p : Partition - m.to - m.from | p.port.messages.t' = p.port.messages.t
  }
}

```

Figure 1: A PIFP model in Alloy.

at most one partition. Fact `PifpOk` uses the cartesian product operator to ensure referential integrity, that is, relation `pifp` only mentions partitions running in the kernel. Alloy supports other structural features not included in this example, namely signature inheritance and relation overloading.

In the local state idiom an operation `op` can be specified using a predicate `pred op[... , t, t' : Time] { ... }` with two special parameters `t` and `t'` denoting, respectively, the pre- and post-state. This model declares two partition operations (`send` and `receive` messages) and one kernel operation (`transfer` messages between partitions). In the body of an operation, to access the pre-state or post-state of a mutable relation it suffices to compose it with `t` or `t'`, respectively.

In the body of operations, constraints that do not refer to `t'` can be seen as

pre-conditions. Otherwise we have post-conditions. For example, expression

```
i.port.messages.t' = i.port.messages.t + m
```

states that, after executing operation `send`, the channel of partition `i` should have an additional message `m`. Notice that frame conditions, stating which mutable relations remain unchanged, should be stated explicitly in each operation.

3 Specifying reactive systems in Alloy

To specify temporal properties we must first model execution traces. A typical Alloy idiom for representing finite prefixes of execution traces is to impose a total ordering on signature `Time` and force that every pair of consecutive states is related by one of the operations. The pre-defined module `util/ordering` can be used for the first constraint:

```
open util/ordering[Time]
```

Among others, inclusion of this module declares the total order relation `next`, and the unary relations `first` and `last` with the first and last time steps in a trace prefix. We can restrict relation `next` to pairs of states related by one of the operations as follows:

```
fact Trans {
  all t : Time, t' : t.next |
    some i, o : Partition | send [i,o,t,t']
  or
    some o : Partition | receive [o,t,t']
  or
    some k : Kernel | transfer [k,t,t']
}
```

The initial state of the system, where the ports of all partitions are empty, can be constrained as follows:

```
fact Init { no Partition.port.messages.first }
```

A desirable safety assertion for this system states that all partitions only receive messages from authorized peers. It can be defined as follows:

```
assert Safety {
  all t : Time, p : Partition, m : Message {
    m.to = p and m in p.port.messages.t
    =>
    p in m.from.(p.kernel.pifp)
  }
}
```

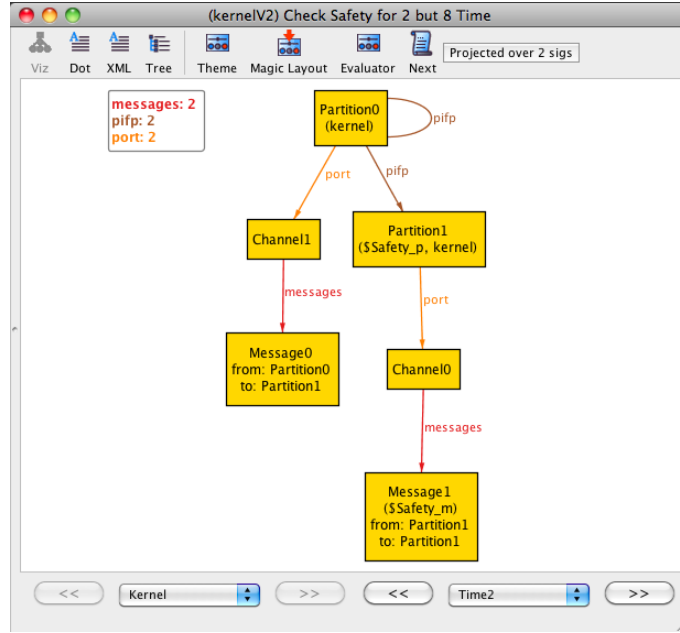


Figure 2: Counter-example.

Unfortunately, model checking this assertion with Alloy Analyzer yields a counter-example. Figure 2 presents the second time step in one of the counter-examples, where we can see that the port of **Partition1** contains a message addressed to itself, which is not authorized by the **pifp** relation. To correct this problem we can, for example, state that all partitions should be allowed to send messages to themselves, by adding the following fact to the model:

```
fact SelfAllow { all p : Partition | p in p.(p.kernel.pifp) }
```

Non-invariant temporal assertions can also be expressed with this idiom, but the complexity of the formulas and expertise required by the modeler increases substantially. Consider, for example the liveness property stating that all authorized messages are eventually transferred to the destination by the kernel. At first glance, it could be specified as follows:

```
assert Liveness {
  all t : Time, p : Partition, m : Message {
    m in p.port.messages.t and m.to in p.(p.kernel.pifp)
    =>
    some t' : t.*next | m in m.to.port.messages.t'
  }
}
```

A simple (but artificial) way to ensure **Liveness** in this model is to disallow message sending while there are still pending messages to be transferred. This

could be done by adding the following pre-condition to operation `send`:

```
no Partition.port.messages.t
```

However, model-checking assertion `Liveness` yields a false counter-example, where a message is sent in the last state of a trace. In fact, if we consider only finite prefixes of execution traces, it is almost always possible to produce a false counter-example to a liveness property. This problem is well-known in the bounded model-checking community, and the solution first proposed in [1] is to only consider as counter-examples to these properties prefixes of traces containing a back loop in the last state, i.e., prefixes that model infinite execution traces. We can add a `loop` state to the `Time` signature and restrict it to the last state as follows:

```
sig Time { loop : lone Time }
fact Loop { loop in last -> Time }
```

Fact `Trans` must also be changed to account for `loop`:

```
fact Trans {
  all t : Time, t' : t.(next+loop) |
    some i, o : Partition | send [i,o,t,t']
  or
  some o : Partition | receive [o,t,t']
  or
  some k : Kernel | transfer [k,t,t']
}
```

Our liveness property can now be correctly specified as follows:

```
assert Liveness {
  all t : Time, p : Partition, m : Message {
    m in p.port.messages.t and m.to in p.(p.kernel.pifp)
    =>
    no loop or some t' : t.*(loop+next) | m in m.to.port.messages.t'
  }
}
```

4 Bounded model checking of LTL formulas in Alloy

As seen in the previous section, although we can specify and model-check temporal properties in standard Alloy, it is a rather tricky and error-prone task. As such, we propose that such properties are expressed using the standard *Linear Temporal Logic* (LTL) without explicitly mentioning time. LTL prescribes the following temporal operators: **X** for next, **G** for always, **F** for eventually, **U** for until, and **R** for release. For example, the previous properties could be specified as follows.

```

assert Safety {
  all p : Partition, m : Message {
    G (m.to = p and m in p.port.messages
      =>
      p in m.from.(p.kernel.pifp)
    )
  }
}

assert Liveness {
  all p : Partition, m : Message {
    G (m in p.port.messages and m.to in p.(p.kernel.pifp)
      =>
      F (m in m.to.port.messages)
    )
  }
}

```

In order to perform bounded model checking of such properties using the Alloy Analyzer, we first must constrain the traces of a model specified according to the local state idiom. This can be done as follows:

1. Change the `Time` signature to include the `loop` relation by re-declaring it as `sig Time { loop : lone Time }` and impose a total ordering on it with `open util/ordering[Time]`.
2. Constrain the transition relation with `fact Trans { all t : Time, t' : t.(next+loop) | ϕ_1 or ... or ϕ_m }`, where for each of the m operation declarations `pred opi [x1 : S1, ..., xn : Sn, t, t' : Time] { ... }` the formula ϕ_i is `some x1 : S1, ..., xn : Sn | opi [x1, ..., xn, t, t']`.
3. Restrict the `loop` relation to the last time instant with `fact Loop { loop in last -> Time }`.

Embedding of LTL formulas into Alloy can be done via an (almost) direct encoding of the translation proposed in [1] for bounded model checking of LTL formulas with a SAT solver. Formally, a formula ϕ occurring in a `fact` or `run` command should be replaced by $\llbracket NNF(\phi) \rrbracket_{\text{init}}$, where $\llbracket \phi \rrbracket_t$ is the embedding function defined in Figure 3, and $NNF(\phi)$ is the well-known transformation that converts formula ϕ to *negation normal form* (where all negations appear only in front of atomic formulas - which in Alloy are typically relational inclusion assertions). The embedding for logic and relational operators is trivial, and thus only a representative subset of Alloy relational logic is presented. A model satisfies `G ϕ` if there is a infinite trace where all states satisfy ϕ . As prescribed in [1], when performing bounded model checking of `G ϕ` , only finite prefix with loops must be considered, otherwise one cannot be sure that property ϕ is not violated further down the trace.

$$\begin{aligned}
\llbracket \mathbf{X} \phi \rrbracket_t &\equiv \text{some } t.(\text{next} + \text{loop}) \text{ and } \llbracket \phi \rrbracket_{t.(\text{next} + \text{loop})} \\
\llbracket \mathbf{G} \phi \rrbracket_t &\equiv \text{some loop and all } t' : t. * (\text{next} + \text{loop}) \mid \llbracket \phi \rrbracket_{t'} \\
\llbracket \mathbf{F} \phi \rrbracket_t &\equiv \text{some } t' : t. * (\text{next} + \text{loop}) \mid \llbracket \phi \rrbracket_{t'} \\
\llbracket \phi \mathbf{U} \psi \rrbracket_t &\equiv \text{some } t' : t. * (\text{next} + \text{loop}) \mid \llbracket \psi \rrbracket_{t'} \text{ and} \\
&\quad \text{all } t'' : t. * (\text{next} + \text{loop}) \ \& \ \wedge (\text{next} + \text{loop}).t' \mid \llbracket \phi \rrbracket_{t''} \\
\llbracket \phi \mathbf{R} \psi \rrbracket_t &\equiv \llbracket \mathbf{G} \psi \rrbracket_t \text{ or some } t' : t. * (\text{next} + \text{loop}) \mid \llbracket \phi \rrbracket_{t'} \text{ and} \\
&\quad \text{all } t'' : t. * (\text{next} + \text{loop}) \ \& \ * (\text{next} + \text{loop}).t' \mid \llbracket \psi \rrbracket_{t''} \\
\llbracket \text{not } \phi \rrbracket_t &\equiv \text{not } \llbracket \phi \rrbracket_t \\
\llbracket \phi \text{ and } \psi \rrbracket_t &\equiv \llbracket \phi \rrbracket_t \text{ and } \llbracket \psi \rrbracket_t \\
\llbracket \phi \text{ or } \psi \rrbracket_t &\equiv \llbracket \phi \rrbracket_t \text{ or } \llbracket \psi \rrbracket_t \\
\llbracket \text{all } x_1 : \Phi_1, \dots, x_n : \Phi_n \mid \phi \rrbracket_t &\equiv \text{all } x_1 : \llbracket \Phi_1 \rrbracket_t, \dots, x_n : \llbracket \Phi_n \rrbracket_t \mid \llbracket \phi \rrbracket_t \\
\llbracket \text{some } x_1 : \Phi_1, \dots, x_n : \Phi_n \mid \phi \rrbracket_t &\equiv \text{some } x_1 : \llbracket \Phi_1 \rrbracket_t, \dots, x_n : \llbracket \Phi_n \rrbracket_t \mid \llbracket \phi \rrbracket_t \\
\llbracket \Phi \text{ in } \Psi \rrbracket_t &\equiv \llbracket \Phi \rrbracket_t \text{ in } \llbracket \Psi \rrbracket_t \\
\llbracket \Phi . \Psi \rrbracket_t &\equiv \llbracket \Phi \rrbracket_t . \llbracket \Psi \rrbracket_t \\
\llbracket \Phi \ \& \ \Psi \rrbracket_t &\equiv \llbracket \Phi \rrbracket_t \ \& \ \llbracket \Psi \rrbracket_t \\
\llbracket \Phi + \Psi \rrbracket_t &\equiv \llbracket \Phi \rrbracket_t + \llbracket \Psi \rrbracket_t \\
\llbracket * \Phi \rrbracket_t &\equiv * \llbracket \Phi \rrbracket_t \\
\llbracket \{x_1 : \Phi_1, \dots, x_n : \Phi_n \mid \phi\} \rrbracket_t &\equiv \{x_1 : \llbracket \Phi_1 \rrbracket_t, \dots, x_n : \llbracket \Phi_n \rrbracket_t \mid \llbracket NNF(\phi) \rrbracket_t\} \\
\llbracket \text{none} \rrbracket &\equiv \text{none} \\
\llbracket \text{univ} \rrbracket &\equiv \text{univ} - \text{Time} \\
\llbracket \text{iden} \rrbracket &\equiv \text{iden} <: (\text{univ} - \text{Time}) \\
\llbracket x \rrbracket_t &\equiv x \\
\llbracket r \rrbracket_t &\equiv \begin{cases} r.t & \text{if } r \text{ mutable} \\ r & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 3: Embedding of temporal facts.

A formula ϕ occurring in an assertion or **check** command should be replaced by **not** $\llbracket NNF(\text{not } \phi) \rrbracket_{\text{init}}$. Notice that, for instance, a counter-example to $\mathbf{F} \phi$ is a trace satisfying its negation, that is $\mathbf{G} (\text{not } \phi)$. Since the Alloy Analyzer negates assertions to find counter-examples we introduce an outer-most negation to cancel it.

Applying this embedding to the temporal formulas presented above yields Alloy formulas equivalent to the ones presented in the previous section. Note that when the **util/ordering** module is imported the scope of the parameter signature is interpreted as an exact scope. This means that trace prefixes are bounded to be of size equal to the scope of the **Time** signature. Thus, to perform bounded model checking of an assertion for a given bound, the user must manually increase the scope of **Time** one unit at a time up to that bound. For example,

```
check Liveness for 3 but 8 Time
```

only performs model checking of assertion **Liveness** for trace prefixes of size exactly 8.

5 Related Work

Several extensions of Alloy to deal with dynamic behavior have been proposed. DynAlloy [4] proposes an Alloy variant that allows the specification of properties over execution traces using a formalism inspired by dynamic logic. Although it simplifies the specification of properties over sequential programs, it is not well suited to deal with concurrency, has a complicated and uncertain semantics, and loses some of the logical nature and expressiveness of Alloy’s language (and appeal). Imperative Alloy [8] proposes a more minimal extension to the language, with a cleaner semantics by means of a simple embedding to standard Alloy. Likewise to DynAlloy it is more geared towards the specification of sequential programs, and since it reuses Alloy’s general purpose verification mechanism it may produce erroneous counter-examples for dynamic assertions (namely, those involving sequential composition of actions).

One of the advantages of the proposed approach is that models can be specified declaratively using Alloy’s relational logic, as opposed to traditional model checkers where state transitions must be specified imperatively. Chang and Jackson [2] proposed a BDD-based model checker for declarative relational models with a syntax very close to Alloy, and where temporal formulas are specified in CTL. The current proposal makes it even easier for Alloy experts to benefit from the power of temporal logic to specify reactive systems, and reuses the very efficient Alloy Analyzer to perform bounded model checking, thus avoiding the maintainability problems associated with the development of a dedicated tool.

Recently, Vakili and Day [9] showed how CTL formulas with fairness constraints can be model checked in Alloy, by using the encoding to first order logic with transitive closure first proposed in [6]. The proposed technique performs full model checking on state transition systems specified declaratively,

but bounded to have at most the number of states specified in the scope. This non-standard form of bounded model checking can yield non-intuitive results and suffers from the state explosion problem that the standard bounded model checking attempts to avoid. Consider for example a model with an integer **counter** that is initially 0 and where each transition increments the **counter** by 1 (modulo the scope of integers, which corresponds to the bit-width used to represent them using two's complement). Given an integer scope of n , bounded model checking of property $\mathbf{F counter} = k$ trivially holds iff the scope of the state is greater or equal then $k + 1$ (assuming $k < 2^{n-1}$). Using the approach proposed in [9] the scope of the state bounds the size of the transition system instead of the size of the trace prefixes. Since according to temporal logic semantics each state is required to always have a successor, that formula will only hold for state scopes greater or equal then 2^n , since no transition system satisfies the specification if the scope is smaller. Moreover, instead of proposing an Alloy extension, CTL formulas are expressed using library functions that compute the set of states where the formula holds. This forces the user to use relational operators to encode logical connectives, instead of the normal logic operators of Alloy.

6 Conclusions

This report proposes an extension to Alloy to specify reactive systems using temporal logic, namely LTL. While retaining the beauty and simplicity of Alloy's relational logic to model declaratively a reactive system, this approach makes it easier to specify its dynamic properties. Moreover, this extension can be embedded in standard Alloy, enabling safe bounded model checking of assertions to be performed with the existing Alloy Analyser. We intend to deploy the proposed extension as a simple command line tool that performs the specified encoding.

References

- [1] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.
- [2] Felix Sheng-Ho Chang and Daniel Jackson. Symbolic model checking of declarative relational models. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 312–320. ACM, 2006.
- [3] Information Assurance Directorate. U.S. government protection profile for separation kernels in environments requiring high robustness, June 2007. Version 1.03.

- [4] Marcelo F. Frias, Juan P. Galeotti, Carlos López Pombo, and Nazareno Aguirre. DynAlloy: upgrading alloy with actions. In Gruiia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 442–451. ACM, 2005.
- [5] Ana Gabriela Garis, Alcino Cunha, and Daniel Riesco. Translating Alloy specifications to UML class diagrams annotated with OCL. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *SEFM*, volume 7041 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2011.
- [6] Neil Immerman and Moshe Y. Vardi. Model checking and transitive-closure logic. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 1997.
- [7] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.
- [8] Joseph P. Near and Daniel Jackson. An imperative extension to Alloy. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *ASM*, volume 5977 of *Lecture Notes in Computer Science*, pages 118–131. Springer, 2010.
- [9] Amirhossein Vakili and Nancy A. Day. Temporal logic model checking in Alloy. In John Derrick, John A. Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *ABZ*, volume 7316 of *Lecture Notes in Computer Science*, pages 150–163. Springer, 2012.